ITOS STOL

ITOS User's Manual

$Date: 2006/06/22 21:02:52 $

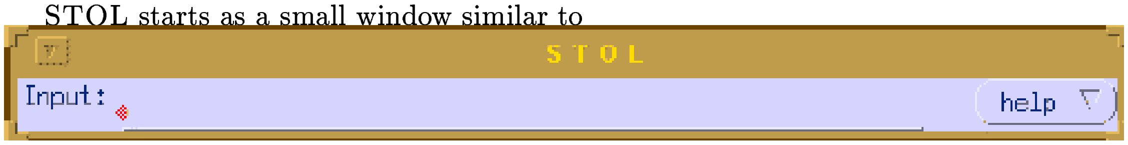
ITOS STOL

STOL, the Spacecraft Test and Operations Language, is a primary user interface to the ITOS. The ITOS dialect of STOL allows the ITOS to be controlled interactively or via STOL procs.

In addition to controlling the ITOS, STOL monitors spacecraft telemetry and can send spacecraft commands. In particular, STOL can synchronize spacecraft commands with telemetry so that, for example, a command is not sent until telemetry indicates the previous command completed.

STOL understands all spacecraft commands and telemetry mnemonics in the Telemetry and Command database.

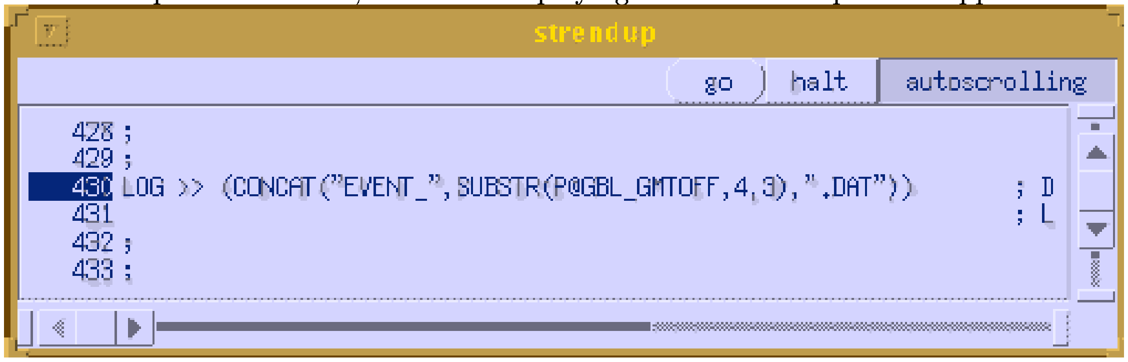
STOL starts as a small window similar to



where the operator may interactively enter directives. There are over 50 directives in STOL, including **`acquire`**, which controls telemetry acquisition; **`cmd`**, which sends spacecraft commands; **`page`** which controls display pages; and **`start`**, which starts procs. See *STOL Directives* for the complete list of STOL directives.

A proc is a file containing directives; procs can be used to save typing or to automate complex tests. A proc is similar to a subroutine in a programming language. STOL provides several directives which only make sense in procs; these include the compound directives **`do while - enddo`** and **`if - elseif - else - endif`** and the synchronization directive **`wait until`**.

When a proc is `started`, a window displaying the text of the proc file appears:



```
;
;
LOG >> (CONCAT("EVENT_",SUBSTR(P@GBL_GMTOFF,4,3),".DAT"))      ; D
                                                               ; L
;
;
```

More than one proc may be open at the same time, but only the most recently opened proc is active – all other open procs are suspended until the active proc completes, and then the youngest of the suspended procs becomes active.

It is possible to control the speed at which the active proc executes, and even to halt the active proc and manually step through its directives one at a time.

1 Constants, Variables, and Expressions

Many directives (`open` and `shoval`, for example) require user-supplied values. These values are either date, float, int, string, time, or unsigned[1].

A *date* represents an absolute GMT time. `TIME42` and `TIME40` mnemonics have date values.

A *float* represents a floating point number. `SFP`, `DFP`, and `DFP085` mnemonics have float values. Also, a polynomial conversion is a float value.

An *int* represents a signed integral value. `SB`, `SI085`, `SLI085`, `SI`, `SI320`, and `SLI` mnemonics have int values.

A *string* represents a string value. `CHAR` mnemonics have string values. Also, a discrete conversion is a string value.

A *time* represents a relative time. `TIME20` and `TIME12` mnemonics have time values.

An *unsigned* represents an unsigned integral value. `USB`, `USI085`, `USLI085`, `USI`, `USI320`, and `USLI` mnemonics have unsigned values.

1.1 Date constants

A date represents an absolute GMT time. A date constant looks like:

```
92-288-14:39:12
 - or -
92-288-14:39:12.0005
```

and consists of a year, day-of-year, hour, minute, second, and (optionally) fractional second. Note that year and day-of-year must be specified in a date constant; otherwise, a time constant is assumed.

1.2 Float constants

A float constant contains a decimal point and/or an exponent. The following are float constants:

```
1.
.2
3.4
5e6
7.8e9
0.1e-2
```

`olstol` stores float values in C language `double` variables.

[1] The `raw` and `rawtf` directives use hexadecimal values – these are described with those directives

1.3 Int constants

An int constant does not contain a decimal point or exponent. The following are int constants:

```
0
1
2048
-93
```

(Strictly speaking, there are no negative int constants. `-93` is really the `-` (unary minus) operator applied to the positive int constant `93`).

`olstol` stores int values in C language `int` variables.

1.4 String constants

Quoted string constants are enclosed in double quotes and may contain blanks or other special characters. Two consecutive double quotes (`""`) inside a quoted string constant represent one double quote in the string. The following are quoted string constants:

```
"string"
"1 + 2 .eq. 3"
"One "" quote"
```

Strings that look like variable or telemetry mnemonic names don't have to be quoted (unless, of course, there is a variable or mnemonic with that name).

In some contexts, string constants may contain special characters even though the string is not enclosed in double quotes.

1.5 Time constants

A time value represents the length of a time interval (i.e, a relative, or delta, time). A time constant looks like:

```
1:23:45
 - or -
0:00:02.5
```

A time constant does not specify a year or day-of-year. `olstol` stores time values in two C language `int` variables; one representing the number of seconds and the other the number of microseconds beyond that number of seconds.

1.6 Unsigned constants

Unsigned constants can be expressed in binary, octal, or hexadecimal. Also, deciaml constants that exceed the maximum signed integer value will automatically be typed as unsigned. The following are unsigned constants:

```
B'1000'
0B1000
O'177'
H'FFFFFFFF'
```

```
X'FFFFFFFF'
0XFFFFFFFF
2147483648
```

1.7 Implicit conversions

The implicit conversions allow, for example, an int constant to be specified when a float value is needed. The following table shows the implicit conversions. Note that no implicit conversion from time to int is listed; instead, there are implicit conversions from time to float and from float to int.

`date`	⇒ string. The resulting string is 22 characters long and has the format `yy-ddd-hh:mm:ss.uuuuuu`.
`float`	⇒ int. The fractional part of the float is trunctated; `22.0`, `22.1`, and `22.9999` all convert to int 22.
	⇒ string. The resulting string will contain a decimal point and at least one digit on either side of the decimal point.
	⇒ time. The float value represents a number of seconds. `123.456` converts to time `00:02:03.456000`.
`int`	⇒ float.
	⇒ string.
	⇒ time. The int value represents a number of seconds.
	⇒ unsigned. Negative int values can be confusing – `-1` converts to unsigned `4294967295`!
`string`	⇒ float.
	⇒ int.
`time`	⇒ date.
	⇒ float. The float represents the number of seconds in the time. Time `12:13:14.5` converts to float `43994.5`.
	⇒ string.
`unsigned`	⇒ int.
	⇒ string.

1.8 Variables

STOL uses two kinds of internal variables, local variables and global variables.

A *local* variable is visible only when the proc in which they were declared is active. Local variables are used to temporarily store a value within a proc. Proc parameters are local variables.

A *global* variable is not connected to any particular proc and can be used to communicate between procs.

If a local variable and a global variable have the same name, the local variable obscures the global variable.

Unlike telemetry mnemonics, STOL variables are typeless. A STOL variable can contain a float value at one moment, a time value later on, and an int value after that.

The `local` and `global` directives create variables; the `free` directive destroys variables. Newly created variables don't have any value and must be assigned before they can be referenced.

1.9 Mnemonics

Telemetry mnemonics are similar to variables and may be used as variables, but unlike variables, each mnemonic has a fixed type. For example, a F1234 mnemonic always has a float value, and cannot store any other type of value. When STOL assign a value to a telemetry mnemonic, it follows the normal implicit conversion rules to obtain a value with a type corresponding to the mnemonic's type.

STOL can assign a value to a mnemonic, but most telemetry mnemonic values are updated frequently by the telemetry subsystem, which overwrites any value assigned by STOL.

When ITOS unpacks a telemetry value, it automatically converts it to the format native to the machine on which ITOS is running. So, if a spacecraft is using a bit-endian processor such as a PowerPC, and ITOS is running on a little-endian machine such as an Opteron, ITOS will convert all telemetry values to their little-endian representation. In most cases this is unimportant to and unnoticed by the user, but there are rare circumstances where it might be confusing. (So, now you know.)

Furthermore, for date mnemonics, ITOS shifts the epoch to the operating system's native epoch. For UNIX systems (which are the only systems on which ITOS runs), that epoch is 70-001-00:00:00.065536. **This is a routine source of much confusion!** See below for more information on times and dates

1.9.1 Dates in STOL

Dates in STOL are all relative to the UNIX epoch regardless of their origin on the spacecraft or elsewhere.

1.10 Command fields

Like telemetry mnemonics, each spacecraft command field has a fixed type. For example, a TIME12 mnemonic always has a time value, and cannot store any other type of value. When sending a spacecraft command, STOL follows the normal implicit conversion rules to assign a value with a the command field's type to the command field.

1.11 Expressions

STOL values may be specified as expressions using arithmetic or logical operators and/or built-in functions. Parenthesis may be used to control the order of evaluation in expressions.

This document describes the arithmetic functions that STOL understands; see *STOL Functions* for the complete list of built-in functions.

1.11.1 Unary +, -, and ~

The unary arithmetic operators take one argument. If this argument is float (or a string which can be converted to float) the result is float; otherwise the result is int.

Unary `+` does nothing except convert is argument to either float or int, and is provided mainly for symmetry with unary `-`.

Unary `~` inverts the bits of an unsigned value.

1.11.2 + – Addition

If one argument is a date and the other argument can be converted to a time, the result of addition will be a date.

Else, if one argument is a time and the other argument can be converted to a time, the result of addition will be a time.

Else, if one argument is float (or a string which can be converted to float) and the other argument can be converted to float, the result of addition will be a float.

Else, if one argument is int (or a string which can be converted to int) and the other argument can be converted to int, the result of addition will be int.

Else, if both arguments are unsigned, the result of addition will be unsigned.

Otherwise, addition cannot be performed and an operator error will occur.

1.11.3 - – Subtraction

If both arguments are date, the result of subtraction will be a time.

Else, if one argument is type date and the other argument can be converted to type time, the result of subtraction will be type date.

Else, if one argument is type time and the other argument can be converted to type type, the result of subtraction will be type time.

Else, if one argument is type float (or type string which can be converted to type float) and the other argument can be converted to type float, the result of subtraction will be type float.

Else, if one argument is type int (or type string which can be converted to type int) and the other argument can be converted to type int, the result of subtraction will be type int.

Else, if both arguments are type unsigned, the result of subtraction will be type int if the second argument is `.GT.` the first argument or type unsigned if the first argument is `.GE.` the second argument.

Otherwise, subtraction cannot be performed and an operator error will occur.

1.11.4 * – Multiplication

If either argument is type date or type time, multiplication cannot be performed and an operator error will occur.

Else, if one argument is type float (or type string which can be converted to type float) and the other argument can be converted to type float, the result of multiplication will be type float.

Else, if one argument is type int (or type string which can be converted to type int) and the other argument can be converted to type int, the result of multiplication will be type int.

Else, if both arguments are type unsigned, the result of multiplication will be type unsigned

Otherwise, multiplication cannot be performed and an operator error will occur.

1.11.5 / – Division

If either argument is type date or type time, or if the second argument can be converted to type int or type float 0, division cannot be performed and an operator error will occur.

Else, if one argument is type float (or type string which can be converted to type float) and the other argument can be converted to type float, the result of division will be type float.

Else, if one argument is type int (or type string which can be converted to type int) and the other argument can be converted to type int, the result of division will be type int.

Else, if both arguments are type unsigned, the result of division will be type unsigned

Otherwise, division cannot be performed and an operator error will occur.

1.11.6 ^ – exponentiation

```
SHOVAL 2 ^ 4 ⇒ 16
SHOVAL .3 ^ 1.5 ⇒ 0.164317
SHOVAL 2 ^ -2 ⇒ 0.25
```

The result of exponentiation is float.

The left arg cannot be negative.

`x^0` ⇒ `0` for all `x`.

1.11.7 .EQ., .NE., .GT., .GE., .LT., and .LE.

The relational operators are `.EQ.`, `.NE.`, `.GE.`, `.GT.`, `.LE.`, and `.LT.`. These binary operators compare their arguments and evaluate to either `0` (false) or `1` (true).

If both arguments are the same type, the comparison is made.

Else if either argument is type string which cannot be converted to either type int or type float, the other argument is converted to type string and the comparison is made.

Else if either argument is type date, the other argument is converted to type date and the comparison is made.

Else if either argument is type time, the other argument is converted to type time and the comparison is made.

Else if either argument is type float, the other argument is converted to type float and the comparison is made.

Else if either argument is type int, the other argument is converted to type int and the comparison is made.

Else both arguments are converted to type unsigned and the comparison is made.

1.11.8 .NOT., .AND., and .OR.

The logical operators are `.NOT.`, `.AND.`, `.OR.`, and `.XOR.`. These operators evaluate to either `0` (false) or `1` (true).

`.NOT.` is a unary operator.

`.AND.` and `.OR.` evaluate left to right and stop evaluating as soon as the result is known. For example, if `x` is 1, then `x .OR. y` does not evaluate `y` since `y`'s value can't affect the expression's value. This can be used to avoid operator errors, as in `y .NE. 0 .AND. x/y .LT. 1`.

1.11.9 Precedence

The precedence of operators is (from highest to lowest):

```
unary -, ~, .NOT.

^

*, /

+, -

.EQ., .NE., .GE., .GT., .LE., .LT.

.AND.

.OR., .XOR.
```

All operators except `^` evaluate left to right. (`^` evaluates right to left – `4^3^2` is 262144 and not 4096). Parenthesis may be used to change the order of evaluation.

2 Proc Files

A proc file is a list of STOL directives, and acts like a subroutine in programming languages. The first directive in a proc file should be a `proc` directive (see ⟨undefined⟩ [PROC], page ⟨undefined⟩) and the last directive should be an `endproc`.

2.1 starting a proc

The `start` directive (see ⟨undefined⟩ [START], page ⟨undefined⟩) starts a proc. This involves several steps:

First, the proc file is located. This is easy when the `IN path` clause is specified in the `START` directive. Otherwise, each directory in the procpath is searched until a file named *procname*`.proc`, where *procname* is the name specified in the `START` directive, is found. The search is case insensitive[1], so `START myproc` could locate `myproc.proc`, `MYPROC.PROC`, or even `MyPrOc.PrOc`!

Once the proc file has been located, a proc window is opened, the file is read into the proc window, and the file is searched for the `PROC` directive, which must be the first directive in the proc (blank lines and comment lines may precede the `PROC` directive, however). The `START` and `PROC` are compared to make sure the number of arguments specified in the `START` matches the number of parameters in the `PROC`.

Each parameter in the `PROC` becomes a local variable, initialized with the corresponding argument value.

If the `AT line` clause is specified in the `START`, a `GOTO line` is performed.

And finally, if the `halted` clause is specified in the `start` directive, the proc is halted. Otherwise the proc executes at the default proc speed (see ⟨undefined⟩ [SPEED], page ⟨undefined⟩).

If errors are detected during startup (and the proc file was found and could be opened), the proc window remains open but the proc is stopped.

Global variables may be used to allow procs to return values.

2.2 proc modes

A proc can be suspended, halted, and/or waiting. (Any combination is allowed). The control bar at the top of each proc window indicates which combination of modes is in effect.

2.2.1 suspended

Whenever more than one proc is running, the most-recently-started proc is the current proc, and all other procs are suspended. The `halt/resume` and `step/go` buttons are stipled when the proc is suspended.

[1] Because the search is case insensitive, it is possible for more than one file in a directory to satisfy the search. This can lead to confusion since it's impossible to predict which file will be found.

2.2.2 halted

A proc is halted whenever the `halt/resume` button is depressed and labeled `resume` (normally, this button is labeled `halt`). When a proc is halted (and not also suspended or waiting), clicking the `step` button executes the next directive in the proc.

A proc may be started halted (i.e., `START MYPROC HALTED`) or may be interactively halted by clicking the button labeled `halt`. Clicking the button labeled `resume` brings the proc out of halt mode.

2.2.3 waiting

A proc is waiting whenever the `step/go` button is labeled `go` (normally, this button is labeled `step`).

A proc enters a wait via a `WAIT` directive, whenever a subsystem needs to temporarily suspend the proc, or as the result of a stol error.

If the wait is the result of a `WAIT UNTIL` directive, the wait terminates when the condition becomes true. The wait terminates even if the proc is also halted and/or suspended when the condition becomes true.

If the wait is the result of a `WAIT interval` directive, the wait terminates when the interval expires. The wait terminates even if the proc is also halted and/or suspended when the interval expires.

If the wait is caused by a subsystem that needs to temporarily suspend the proc, the wait terminates when the subsystem allows the proc to continue. The wait terminates even if the proc is also halted and/or suspended whent the subsystem allows the proc to continue.

And regardless of how the wait was caused, the wait terminates when the button labeled `go` is clicked or a `GO` directive is entered.

2.3 proc control directives

Several directives are only useful in proc files. These are:

`PROC` `ENDPROC`	These are the first and last directives in every proc file.
`LOCAL`	This directive creates local variables.

`IF`
`ELSEIF`
`ELSE`
`ENDIF`
	These directives control conditional execution.

`DO`
`ENDDO`
`BREAK`
`CONTINUE`
	These directives control `DO WHILE` and `DO UNTIL` loops.

2.3.1 labels

Proc statements may be labeled so they can be the destination of a `goto` directive (see ⟨undefined⟩ [GOTO], page ⟨undefined⟩). Some examples of labels are:

```
PROC myproc
shoval "here comes a label"
label1:
shoval "another label"
label2: shoval "here it is"
ENDPROC
```

A label has a maximum length of 31 characters. Longer labels will be truncated and cause the GOTO directive to not find a match and fail.

2.4 the procpath

See *GBL_PROCPATH*.

2.5 proc examples

bullet STOL Procs that included with ITOS.

3 The Interactive Window

The interactive window is the only STOL window that is always displayed. The interactive window allows the Test Conductor to interactively enter STOL directives.

3.1 Control keys

Some of the edit commands available in the interactive window are:

`^B`
`(left arrow)`
: Move the cursor one space left.

`^F`
`(right arrow)`
: Move the cursor one space right.

`^P`
`(up arrow)`
: Display the previous command in the history list. At least 25 commands are remembered.

`^N`
`(down arrow)`
: (When viewing a previous command) display the next command in the history list.

`^A` Move the cursor to the beginning of the line.

`^E` Move the cursor to the end of the line.

`^H`
`(Back Space)`
: Delete the character before the cursor.

`^D` Delete the character at the cursor.

`^K` Delete all characters from the cursor to the end of the line.

3.2 Help

Clicking the `help` button brings up the help window, and displays the latest version of this manual.

4 Proc File Windows

Each proc file executes in a seperate window. Clicking the mouse at various places in these windows affects how the proc executes.

4.1 The autoscrolling/using scrollbar toggle

Normally a proc window is *autoscrolling*, which means that the view will automatically scroll so the directive being executed is displayed.

Toggling the `autoscrolling` button changes the button's label to `using scrollbar` and stops the automatic scrolling so the scrollbars can be used to view other sections of the proc.

4.2 The halt/resume toggle

The button labeled either `halt` or `resume` controls whether or not the proc is halted. When labeled `halt`, clicking this button halts the proc and changes the label to `resume`; when labeled `resume`, clicking this button resumes the proc and changes the label to `halt`.

4.3 The step/go button

The button labeled either `step` or `go` either restarts a waiting proc or executes the next directive in a halted proc.

4.4 Mouse GOTO

To `goto` using the mouse, move the mouse to the line you want to go to, hold down the control key, and click the left button.

Table of Contents